APEX/HPX Integration Specification for Phylanx

Kevin A. Huck (PI), Allen D. Malony, and Monil Mohammad Alaul Haque

Oregon Advanced Computing Institute for Science and Society (OACISS)
at the University of Oregon,
in collaboration with Louisiana State University and
the University of Arizona

March 1, 2019

Project:	Phylanx
Project Title:	An Asynchronous Distributed C++ Array Processing Toolkit
Period:	1 July 2018 – 28 February 2019
PI:	Kevin A. Huck
Contract Number:	FA8075-14-D-0002-0007, TAT 15-1158
Subcontract Number:	S901070BAH
Project Number:	09003-0725
Institute:	University of Oregon Oregon Advanced Computing Institute for Science and Society 470 Streisinger Hall 5294 University Of Oregon Eugene OR 97403-5294

Contents

1 Introduction

This document summarizes and describes how APEX is integrated into the Phylanx project. Specifically, this document describes how it integrates with HPX – a key dependency of the Phylanx ecosystem.

The APEX (Autonomic Performance Environment for eXascale) library provides a novel approach to performance observation, measurement, analysis and runtime decision making in order to optimize performance. The particular challenges of accurately measuring the performance characteristics of HPX applications (as well as other asynchronous multitasking runtime architectures) required a new approach to parallel performance observation. The standard model of multiple operating system processes and threads observing themselves in a first-person manner while writing out performance profiles or traces for offline analysis does not adequately capture the full execution context, nor provide opportunities for runtime adaptation within HPX applications. APEX includes methods for information sharing between the layers of the software stack, from the hardware through operating and runtime systems. The performance measurement components incorporate relevant information across stack layers, with merging of third-person performance observation of node-level and global resources, remote processes, and both operating and runtime system threads. For a complete description of APEX, see [4].

In short, APEX is an introspection and runtime adaptation library for asynchronous multitasking runtime systems. However, APEX is not only useful for AMT/AMR runtimes - it can be used by any application wanting to perform runtime adaptation to deal with heterogeneous and/or variable environments. Used with the HPX runtime, APEX has demonstrated accurate and useful performance measurement at large scales [3], and has been instrumental in performance measurement and adaptation for applications built with HPX.

1.1 Usage in the Phylanx Project

The goal of the Phylanx project is to provide a general purpose solution for computations of distributed arrays for applied statistics on distributed high performance computing resources. Phylanx builds upon Spartan, Theano and TensorFlow in an effort to generalize away low-level operations and support distributed computing. The system will decompose array computations into a predefined set of parallel operations and employ algorithms that optimize execution and data layout based on a user-provided expression graph in the form of a Python application. Using hints and annotations provided by the user and the subsequent optimization steps, the expression graph (or abstract syntax tree, AST) will be passed to the HPX runtime which schedules work and infers the data layout on the provided computing resources. Early results [7] have been promising, and as the Phylanx project evolves, we expect to see significant progress as support is added for large distributed computations on heterogeneous hardware.

Because APEX provides performance measurement and runtime feedback/control of HPX applications, it is a natural fit to assist in providing tuning and optimization support for the Phylanx project. The remainder of this section will describe APEX at a high level, and the remainder of the document will describe both the integration of APEX within the HPX ecosystem and the contributions from the University of Oregon to the Phylanx project during the current project period. We conclude this report with a description of Continuous Integration efforts implemented by the University of Oregon in Section 7.

1.2 APEX Introspection

APEX provides a synchronous API for measuring actions within a runtime. The API includes methods for timer start/stop, as well as sampled counter values. APEX is designed to be integrated into a runtime, library and/or application and provide performance introspection for the purpose of runtime adaptation. While APEX can provide rudimentary post-mortem performance analysis measurement, there are many other performance measurement tools that perform that task much better (such as TAU). That said, APEX includes an event listener that integrates with the TAU measurement system, so APEX events can be forwarded to TAU and collected in a TAU profile and/or trace to be used for post-mortem performance analysis. We describe the integration of APEX into HPX in Sections 2, 3, and 4. We also describe the visualization and analysis efforts specific to the Phylanx project in Section 5.

1.3 APEX Runtime Adaptation

APEX provides a mechanism for dynamic runtime behavior, either for autotuning or adaptation to a changing computation environment. The infrastructure within APEX that provides the adaptation is the *Policy Engine*, executing policies either periodically or triggered by events. The policies have access to the performance state as observed and updated by the APEX introspection API. APEX is integrated with Active Harmony to provide dynamic search for autotuning. We describe the development of new APEX policies for the Phylanx project in Section 6.

2 HPX Integration - Technical details

2.1 Overview

The original APEX implementation was designed to work with multiple asynchronous multitasking runtimes. At the time of its inception, that included HPX (LSU), HPX-5 (Indiana University), C++ `std::async()` threads, OpenMP [1], and others. However, the HPX implementation was unique in that the asynchronous behavior inside APEX was also implemented with HPX. The intention was that the HPX runtime would have control over the asynchronous background behavior within APEX. For that reason, the APEX and HPX code bases are closely tied together, and in fact have circular dependencies. The following subsections describe the manner in which APEX is used in HPX, and HPX is used in APEX.

2.2 APEX code in HPX

For the most part, APEX measurement is integrated into HPX in two ways – APEX timers are integrated into the task scheduler, and APEX counters are integrated into the HPX performance counters (see `https://stellar-group.github.io/hpx/docs/sphinx/latest/html/manual/optimizing_hpx_applications.html` for details on the HPX performance counters).

APEX is initialized and finalized in HPX using a scoped variable technique – the APEX support is contained in a class called `hpx::util::apex_wrapper_init`, as shown on lines 121–133 of Listing 10. The object is constructed during initialization of `hpx::run_or_start()` (see Listing 1). When the `apex` object declared on line 631 goes out of scope, `apex::finalize()` will be called. The implementation of the APEX wrapper classes is provided in Appendix B, in code listings for `apex.hpp` and `apex.hpp`.

```
628     // Setup all internal parameters of the resource_partitioner
        rp.configure_pools();
630
        util::apex_wrapper_init apex(argc, argv);
632
        // Initialize and start the HPX runtime.
```

```
    LPROGRESS_ << "run_local: create runtime";
```

Listing 1: APEX initialization in HPX.

When an HPX *thread* (aka task) is constructed, an `hpx::util::apex_task_wrapper` object is constructed, is stored in the HPX thread using a `std::shared_ptr<apex::task_wrapper>`, and is destroyed when the thread is destroyed, or when APEX has released the shared pointer after asynchronous background processing. When the HPX thread is executed by the thread scheduler, the thread is measured using APEX timers. The APEX timers are wrapped in an object of type `hpx::util::apex_wrapper` as a scoped variable in the `hpx::threads::detail::scheduler_loop()` method, as shown in Listing 2. If an HPX thread is interrupted without termination, then APEX will not increase the counter for the number of times that thread type was executed (i.e. the task timer will be yielded instead of stopped). Regardless, on either a `stop()` or `yield()` call, APEX will take timestamps and provide them to the APEX back end for asynchronous processing.

```
#if defined(HPX_HAVE_APEX)
// get the APEX data pointer, in case we are resuming the
// thread and have to restore any leaf timers from
// direct actions, etc.

// the address of tmp_data is getting stored
// by APEX during this call
    util::apex_wrapper apex_profiler(thrd->get_apex_data());

    thrd_stat = (*thrd)();

    if (thrd_stat.get_previous() == terminated) {
        apex_profiler.stop();
        // just in case, clean up the now dead pointer.
        thrd->set_apex_data(nullptr);
    } else {
        apex_profiler.yield();
    }
#else
    thrd_stat = (*thrd)();
#endif
```

Listing 2: APEX timer in HPX.

HPX counters can be periodically queried by an application, and when they are, APEX will also store the counter value. In `hpx::util::query_counters::print_value()`, the APEX API call `apex::sample_value()` is executed to store the value, as shown in Listing 3.

```
    template <typename Stream>
    void query_counters::print_value(Stream* out, std::string const& name,
        performance_counters::counter_value const& value, std::string
   const& uom)
    {
        error_code ec(lightweight);        // do not throw
        double val = value.get_value<double>(ec);

        if(!ec) {
#ifdef HPX_HAVE_APEX
            apex::sample_value(name.c_str(), val);
```

Listing 3: APEX counter in HPX.

2.3 HPX code in APEX

The APEX measurement library was designed to handle synchronous tasks as quickly and efficiently as possible, to reduce overhead or perturbing the application being measured. For that reason, building a performance profile at runtime is a task that is handled asynchronously, on a background thread. When timers and counters are observed by the APEX API, they are placed on an internal asynchronous lock-free queue for processing. When measuring runtimes other than HPX, the background processing happens with a C++ `std::thread` object. However, in the HPX integration we wanted to schedule the updating of statistics as a background task. In the standard case, a semaphore is used to signal the background thread to wake up and process tasks. In the HPX integration, APEX will randomly (0.1% of the time) fire off an HPX task to process the HPX thread after stopping a timer. This is done to avoid scheduling background work until there is a significant amount of work to be done, to avoid scheduling overhead. The scheduling logic is shown in Listing 4. The HPX action `apex_schedule_process_profiles` is shown in Listing 5.

```
  inline void profiler_listener::push_profiler(int my_tid,
    std::shared_ptr<profiler> &p) {
        // if we aren't processing profiler objects, just return.
        if (!apex_options::process_async_state()) { return; }
#ifdef APEX_TRACE_APEX
        if (p->get_task_id()->name == "apex::process_profiles") { return;
    }
#endif
      thequeue()->enqueue(p);

#ifndef APEX_HAVE_HPX
      // Check to see if the consumer is already running, to avoid calling
      // "post" too frequently - it is rather costly.
      if(!consumer_task_running.test_and_set(memory_order_acq_rel)) {
        queue_signal.post();
      }
#else
      // only fire off an action 0.1% of the time.
      static int thresh = RAND_MAX/1000;
      if (std::rand() < thresh) {
        apex_schedule_process_profiles();
      }
#endif
  }
```

Listing 4: Signaling APEX background processing.

```
#ifdef APEX_HAVE_HPX
HPX_DECLARE_ACTION(
    APEX_TOP_LEVEL_PACKAGE::profiler_listener::process_profiles_wrapper,
    apex_internal_process_profiles_action);
HPX_ACTION_HAS_CRITICAL_PRIORITY(apex_internal_process_profiles_action);
HPX_PLAIN_ACTION(
    APEX_TOP_LEVEL_PACKAGE::profiler_listener::process_profiles_wrapper,
    apex_internal_process_profiles_action);

void apex_schedule_process_profiles() {
    if(get_hpx_runtime_ptr() == nullptr) return;
    if(!thread_instance::is_worker()) return;
    if(hpx_shutdown) {
        APEX_TOP_LEVEL_PACKAGE::profiler_listener::
   process_profiles_wrapper();
    } else {
        if(!consumer_task_running.test_and_set(memory_order_acq_rel)) {
            apex_internal_process_profiles_action act;
            try {
                hpx::apply(act, hpx::find_here());
            } catch(...) {
                // During shutdown, we can't schedule a new task,
                // so we process profiles ourselves.
                profiler_listener::process_profiles_wrapper();
            }
        }
    }
}
```

Listing 5: Scheduling APEX background processing with an HPX action.

2.4 OTF2, TAU integration with APEX

The APEX synchronous API implements an event listener design. When timers are started and stopped, and when counters are sampled, those events are passed on to additional listener objects that can perform optional functionality within APEX. Two such items are OTF2 trace processing and TAU profiling and/or tracing support. The OTF2 listener object is used to interact with the OTF2 library API and generate an event trace for post-processing. The TAU listener is similar, it is used to interact with the TAU measurement library and generate either a TAU profile or trace for post-processing. In this way, APEX operates as instrumentation glue between asynchronous tasking runtimes and measurement libraries that are not currently capable of measuring asynchronous, pre-emptive runtimes. See the Sections 3 and 4 for details on how to utilize these integrated measurement libraries. To use TAU integration at runtime, the `tau_exec` wrapper script is used to preload TAU libraries. For example, to launch the Phylanx LRA example with TAU support, see Listing 6. In this example, TAU will generate a profile from the APEX timer calls, and will also enable *event based sampling* to measure code executed outside of HPX threads. For more details on using `tau_exec`, see the TAU documentation at http://tau.uoregon.edu.

```
export APEX_TAU=1
tau_exec -T pthread -ebs ./lra_csv
```

Listing 6: Using `tau_exec` to add TAU measurement support to APEX.

3 Relevant APEX Configuration Options

The full configuration and build process for the Phylanx and HPX projects are beyond the scope of this document. For up-to-date instructions, please refer to the Phylanx Github.com project repository (https://github.com/STEllAR-GROUP/phylanx/wiki/Build-Instructions). To enable APEX support in the Phylanx project, HPX needs to be configured and built with APEX. For example, if the regular HPX CMake configuration commands are those shown in Listing 7, adding APEX support is as simple as Listing 8. For more advanced (and useful) support, the full complement of APEX options is shown in Listing 9.

```
cmake \
 -DCMAKE_CXX_COMPILER=/usr/bin/clang++ \
 -DCMAKE_C_COMPILER=/usr/bin/clang -DCMAKE_BUILD_TYPE=Release \
 -DBOOST_ROOT=/opt/local -DHPX_WITH_MALLOC=system \
 -DHWLOC_ROOT=/opt/local -DCMAKE_INSTALL_PREFIX=/opt/local/hpx \
 /Users/user/src/hpx
```

Listing 7: HPX configuration without APEX.

```
cmake \
 -DCMAKE_CXX_COMPILER=/usr/bin/clang++ \
 -DCMAKE_C_COMPILER=/usr/bin/clang -DCMAKE_BUILD_TYPE=Release \
 -DBOOST_ROOT=/opt/local -DHPX_WITH_MALLOC=system \
 -DHWLOC_ROOT=/opt/local -DCMAKE_INSTALL_PREFIX=/opt/local/hpx \
 -DHPX_WITH_APEX=TRUE \
 /Users/user/src/hpx
```

Listing 8: Simple HPX configuration with APEX.

```
cmake \
 -DCMAKE_CXX_COMPILER=/usr/bin/clang++ \
 -DCMAKE_C_COMPILER=/usr/bin/clang -DCMAKE_BUILD_TYPE=Release \
 -DBOOST_ROOT=/opt/local -DHPX_WITH_MALLOC=system \
 -DHWLOC_ROOT=/opt/local -DCMAKE_INSTALL_PREFIX=/opt/local/hpx \
 -DHPX_WITH_APEX=TRUE \
 -DAPEX_WITH_ACTIVEHARMONY=TRUE \
 -DACTIVEHARMONY_ROOT=/opt/local/activeharmony/4.6 \
 -DAPEX_WITH_OTF2=TRUE \
 -DOTF2_ROOT=/opt/local/otf2/2.1 \
 -DAPEX_WITH_PAPI=TRUE \
 -DPAPI_ROOT=/opt/local/papi/5.6.0 \
 /Users/user/src/hpx
```

Listing 9: Complete HPX configuration with APEX.

The HPX configuration process will automatically clone the APEX code repository and include it in the HPX code base. APEX will be built as a dependent library of the HPX library. The APEX configuration options are:

- `HPX_WITH_APEX` (default:FALSE) Include APEX support in HPX
- `APEX_WITH_ACTIVEHARMONY` (default:FALSE) Include Active Harmony policy/tuning support in APEX

- `ACTIVEHARMONY_ROOT` Path to Active Harmony installation
- `APEX_WITH_OTF2` (default:FALSE) Include OTF2 trace library support in APEX
- `OTF2_ROOT` Path to OTF2 installation
- `APEX_WITH_PAPI` (default:FALSE) Include PAPI hardware counter support in APEX
- `PAPI_ROOT` Path to OTF2 installation

4 Relevant APEX Runtime Options

The APEX library has a large number of runtime options, however only a few of them are relevant for the Phylanx project. The runtime options of interest are listed here. To set them, set them as environment variables before running the Phylanx application.

- `APEX_DISABLE` (default:0) Enable/Disable the APEX library entirely. When disabled, APEX will not be initialized, no measurements will be made, no output given. All APEX API calls will return immediately.
- `APEX_PROCESS_ASYNC_STATE` (default:1) Disable the APEX profile statistics computation, to reduce overhead. Useful when collecting a trace without a profile, or when passing events on to TAU.
- `APEX_TAU` (default:0) Enable/Disable the TAU integration.
- `APEX_OTF2` (default:0) Enable/Disable the OTF2 trace collection.
- `APEX_POLICY` (default:1) Enable/disable the APEX policies.
- `APEX_SCREEN_OUTPUT` (default:0) Enable/disable a report of HPX tasks measured by APEX.
- `APEX_TASKGRAPH_OUTPUT` (default:0) Enable/disable the collection of a Graphviz (Dot) style taskgraph of HPX/Phylanx task types.
- `APEX_PAPI_METRICS` (default:) Enable additional PAPI hardware counter metrics
- `APEX_OUTPUT_FILE_PATH` (default:./) The location where output files will be written
- `APEX_OTF2_ARCHIVE_PATH` (default:OTF2_archive) The location where OTF2 traces will be written
- `APEX_OTF2_ARCHIVE_NAME` (default:APEX) The name of the OTF2 file.

5 Traveler-Gantt Visualization & Analysis

Our Phylanx collaborators at the University of Arizona have been developing the *Traveler-Gantt* tool for trace visualization and analysis. The Traveler-Gantt tool is a web-based redesign of Ravel [6], a tool designed for MPI applications and extended to Charm++ applications [5]. Traveler-Gantt is one of the project outcomes for the Phylanx project.

APEX measures Phylanx applications using the integrated HPX instrumentation. Using the available OTF2 trace file output support, APEX can generate trace files that can be interactively explored with Traveler-Gantt. Figure 1 shows a Phylanx trace loaded into the Traveler-Gantt user interface. A longer discussion of Traveler-Gantt will be provided by the Arizona team in their reporting, as well as in the common Phylanx reports from all three Universities. Briefly, the user interface includes a full trace timeline across the bottom of the window, and a zoomed-in region showing task dependencies in the majority of the window.

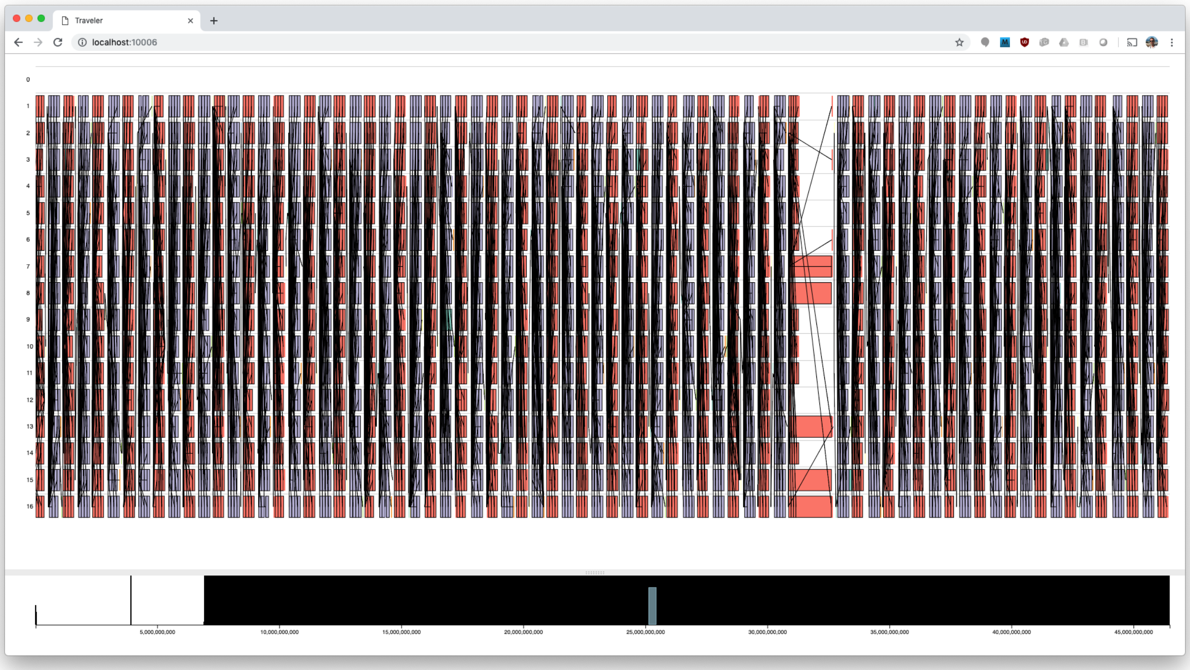


Figure 1: An APEX trace of the LRA Phylanx application visualized in Traveler-Gantt.

5.1 Critical Path Analysis

One key goal for the Traveler-Gantt effort is to provide a measure of critical path analysis for Phylanx applications. This analysis is needed to help find and determine the cause of regions of low concurrency in Phylanx applications. As a requirement for that effort, APEX was extended to generate globaly unique identifiers (GUIDs) for all tasks in the execution. When an HPX task is constructed, its GUID and parent GUID are recorded so that the trace contains the task dependency relationships for all tasks. The Traveler-Gantt application then visually links the tasks with black lines. Figure 2 shows the user interface view when a task is selected in Traveler-Gantt – all of the parent tasks leading up to that task are highlighted with lines of increasing thickness as the dependency chain goes deeper.

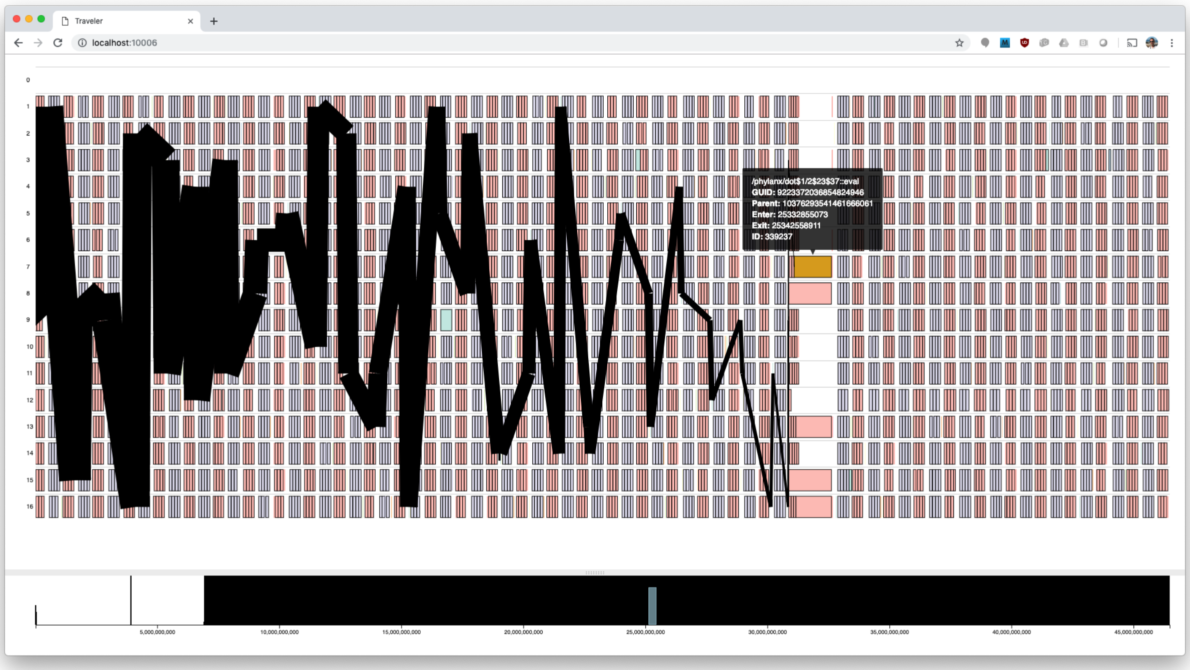
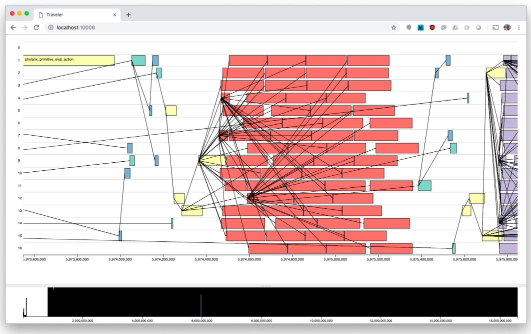

```
/phylanx/dot$1/2$23$37::eval
GUID: 9223372036854824946
Parent: 1037629354146166061
Enter: 25332855073
Exit: 25342558911
ID: 339237
```

Figure 2: Critical path visualization in Traveler-Gantt. When the Phylanx primitive task representing a `dot product` is selected, its color changes to gold and the task hierarchy showing tasks dependent on that task are highlighted.

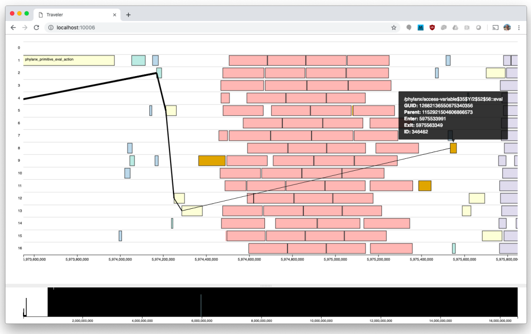
Showing another example in finer detail, the subfigures in Figure 3 Show different states of the Traveler-Gantt GUI when interacting with the APEX data. Subfigure 3a shows a zoomed-in region of the ALS algorithm implemented in Phylanx. After clicking on one task once 3b, the task details are shown, its color changes to gold, and the parent dependency chain is highlighted. Clicking on the task a second time shows all sibling tasks for all parents in the dependency chain 3c. Clicking on the task a third time returns all of the parent-child relationship lines to the GUI 3d. Traveler-Gantt shows great promise in helping to understand critical path dependencies between tasks, and future work should allow us to automatically determine the causes of low concurrency in HPX.

6 APEX Policies

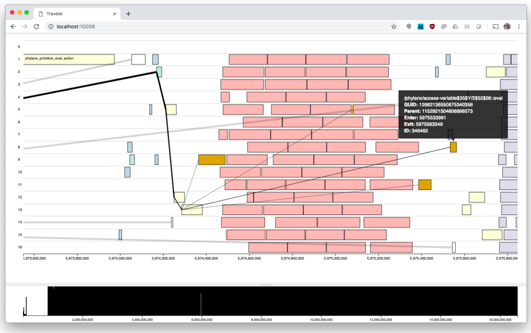
During this phase of the Phylanx project, the University of Oregon team has worked with the Louisiana State University team to investigate polices to help with runtime optimization of parameters in the HPX runtime or the Phylanx library. Two different policies are described below.



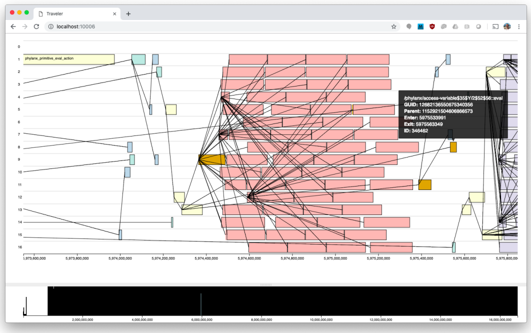
(a) Zoomed in region of the ALS application.



(b) A Phylanx primitive is selected, showing the previous pre-empted instances of the task in gold as well as the tasks that depend on the result of this task.



(c) The primitive is selected with a second mouse click, indicating the sibling tasks of all parent tasks in the dependency chain.



(d) Clicking the task a third time restores all task dependency relationships in the GUI.

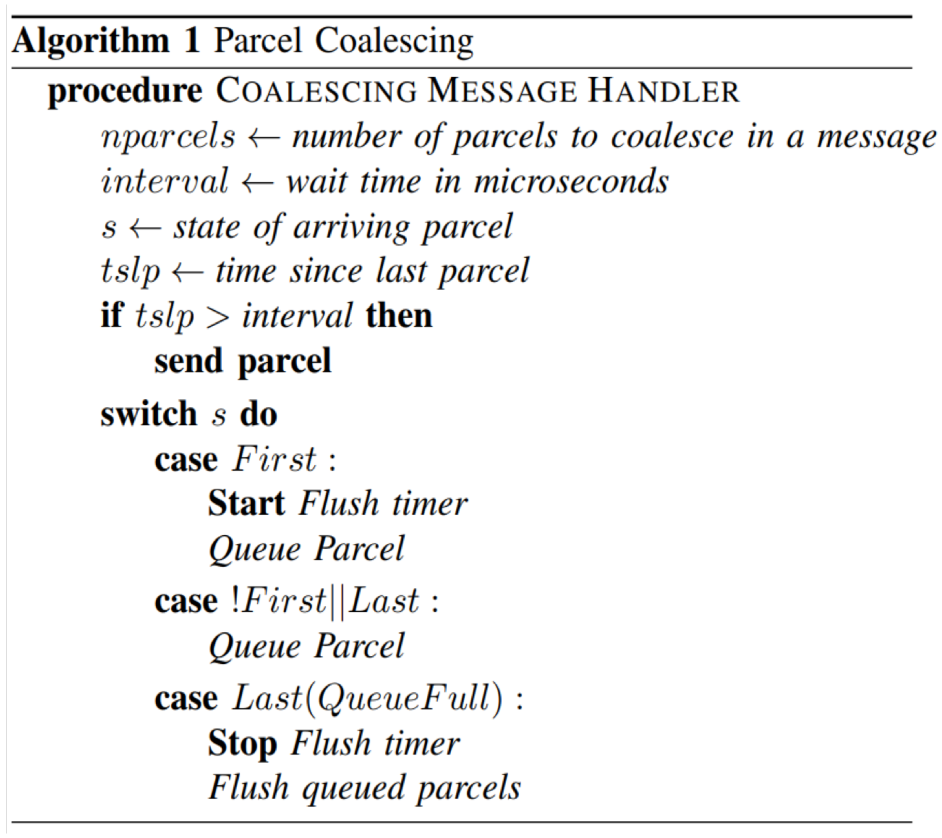
Figure 3: ALS Critical Path visualization in the Traveler-Gantt GUI.

6.1 Parcel Coalescing

Lightweight tasks in distributed HPX applications produce high volumes of fine-grained communication. HPX uses a *Parcel Coalescing* technique, as shown in Figure 4, to reduce communication overhead. The technique depends upon two parameters, the number of parcels to accumulate before transmission and a timeout interval value if that number of parcels is not reached. The number of parcels to accumulate and the coalescing interval have a significant impact on performance. Recent research by the HPX team [8] has demonstrated a positive correlation between task overhead (network overhead) and overall execution time.

Rather than perform an exhaustive search for the best parameter values, we defined a Parcel Coalescing Policy with the option to trigger the policy periodically or based on an event (for example, every 5000 messages). It can start with a default, random, or user provided starting values for the interval and the number of messages to coalesce.

The callback function for the policy is a call to Active Harmony with the APEX sampled counter value of network overhead (as the dependent variable) and the current values of the timeout interval and number of messages (as independent variables). Active harmony ob-



```
Algorithm 1 Parcel Coalescing
  procedure COALESCING MESSAGE HANDLER
      $nparcels \leftarrow$ number of parcels to coalesce in a message
      $interval \leftarrow$ wait time in microseconds
      $s \leftarrow$ state of arriving parcel
      $tslp \leftarrow$ time since last parcel
      if $tslp > interval$ then
          send parcel
      switch $s$ do
          case $First$ :
              Start Flush timer
              Queue Parcel
          case $!First||Last$ :
              Queue Parcel
          case $Last(QueueFull)$ :
              Stop Flush timer
              Flush queued parcels
```

Figure 4: The parcel coalescing algorithm in HPX.

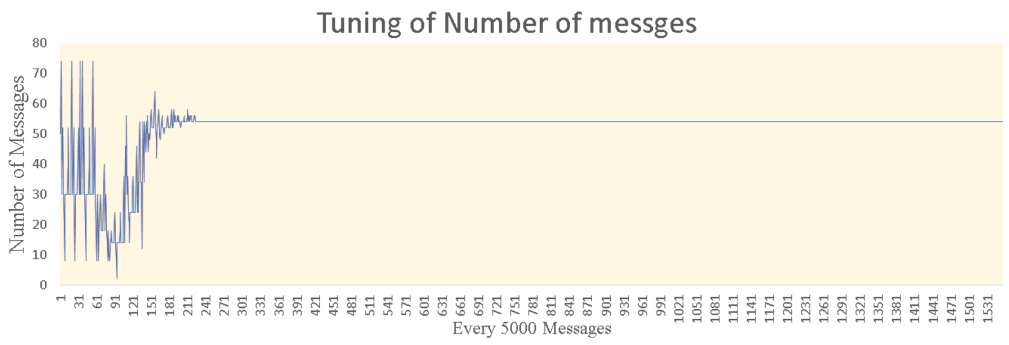
serves the dependent variable to change the value of the two independent variables, searching for the minimal dependent value.

The subfigures in Figure 5 represent the impact of the policy on a synthetic benchmark where the policy is triggered every 5000 message send events. It shows the convergence of the two coalescing parameters and the subsequent reduction in network overhead.

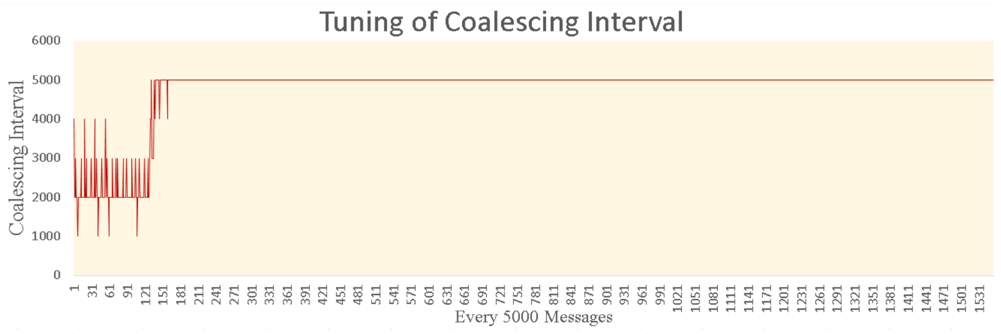
6.2 Direct Actions

In task-based runtimes, one challenge is to find the minimum amount of work size of work (measured as either number of instructions or time to execute) to decide whether a new task will be generated for the work or the current task will immediately execute the work. Creating more tasks will provide more parallelism but an excessive amount of small tasks will generate scheduler overhead. The challenge is to find the “work size” threshold at runtime as it varies from application-to-application and architecture-to-architecture. For Phylanx the threshold should be defined and tuned for every primitive instance, each of which is represented by a node in the abstract syntax tree (AST). The policy will search for a threshold value for every primitive instance providing optimal performance. Every primitive instance launches its own policy and the policy observes the execution time for that instance and decides an optimal value which will theoretically provide faster execution.

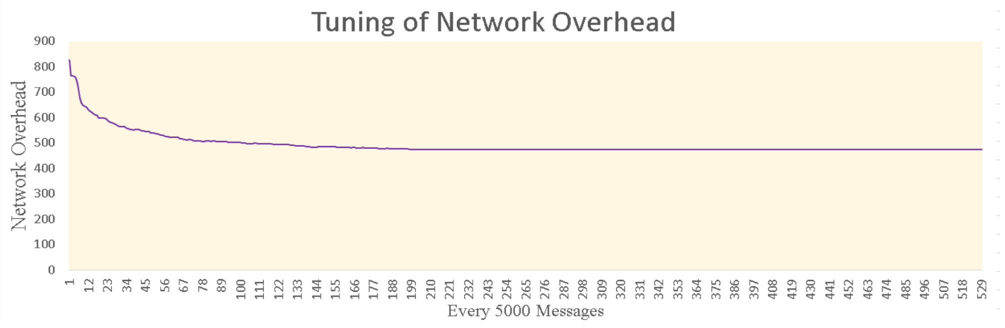
A policy was constructed to search for the optimal threshold for each primitive, driven



(a) Evolution of the number of messages coalescing parameter.



(b) Evolution of the interval coalescing parameter.



(c) Evolution of the observed network overhead counter in HPX.

Figure 5: Effectiveness of the APEX parcel coalescing policy in HPX.

by the time it takes to execute each primitive. If we compare the impact of the policy with naïvely executing all tasks asynchronously, the policy provides roughly a 30-40% improvement. We also compared this adaptive APEX policy with setting a fixed threshold which we call a *basic policy*. We have used the LSU cluster ROSTAM which has Intel Xeon processors for these experiments. Figure 6 compares the basic policy and APEX policy. It shows that in 15 out of 49 cases, the Apex policy outperforms the basic policy. However, the difference between APEX and basic policy is not significant.

We investigated this issue by changing the fixed threshold in the basic policy and observed the changes, as shown in Figure 7. The x-axis of the ALS graph is the lower and upper threshold combination. As the comparison graph suggests, if the lower threshold is higher than 100ms the impact of changing the threshold is not significant. As long as the lower threshold is above 100ms, changing the threshold does not have an effect, and it also suggests

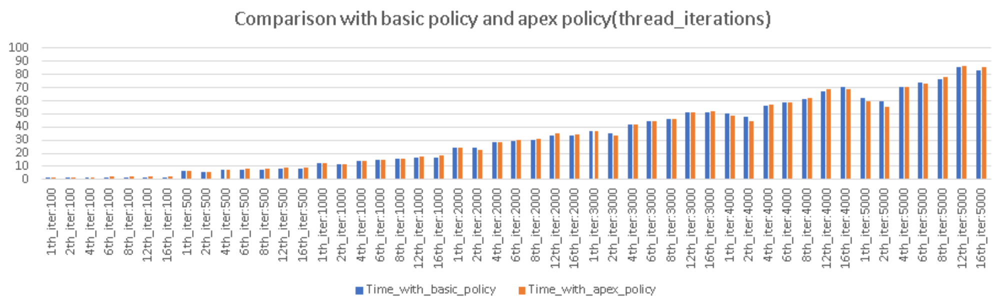


Figure 6: Comparing the basic implementation with the APEX policy.

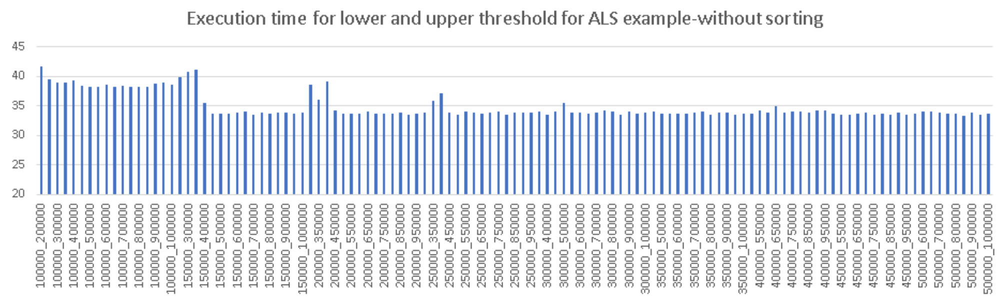


Figure 7: Parametric study of lower and upper threshold values for direct action execution.

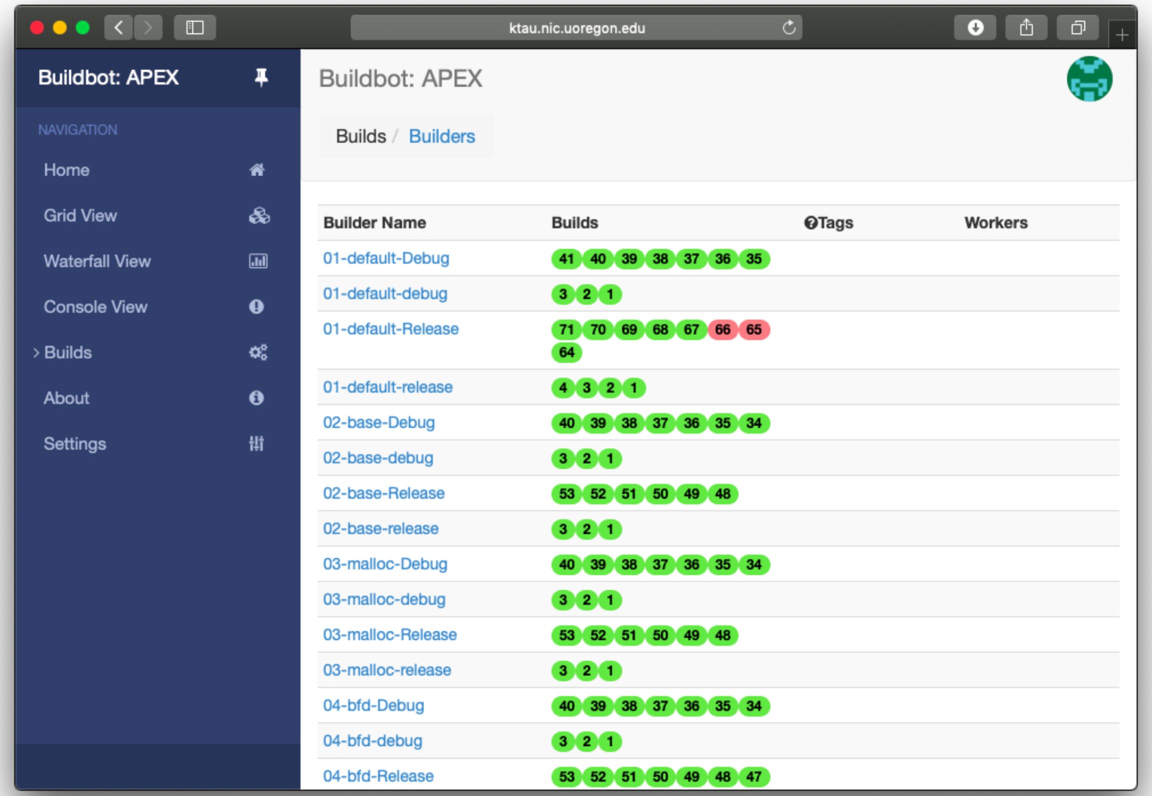
the threshold can be any value without impacting performance significantly. As we expect to see a correlation between the architecture and task size, now we are planning to experiment with other architectures to observe the effect on performance.

7 Continuous Integration Efforts

The Phylanx project uses *continuous integration (CI)* services wherever possible. The main source code repositories for HPX and Phylanx use various automated build-and-test systems to try to prevent the introduction of software errors or regressions of previously fixed bugs. However, for practicality concerns, the main CI solutions enabled only cover a limited set of unit tests. Larger integrated tests for the Phylanx project are necessary, and so at the University of Oregon, we have configured and enabled two CI solutions for both correctness and performance regression testing.

7.1 Buildbot Services

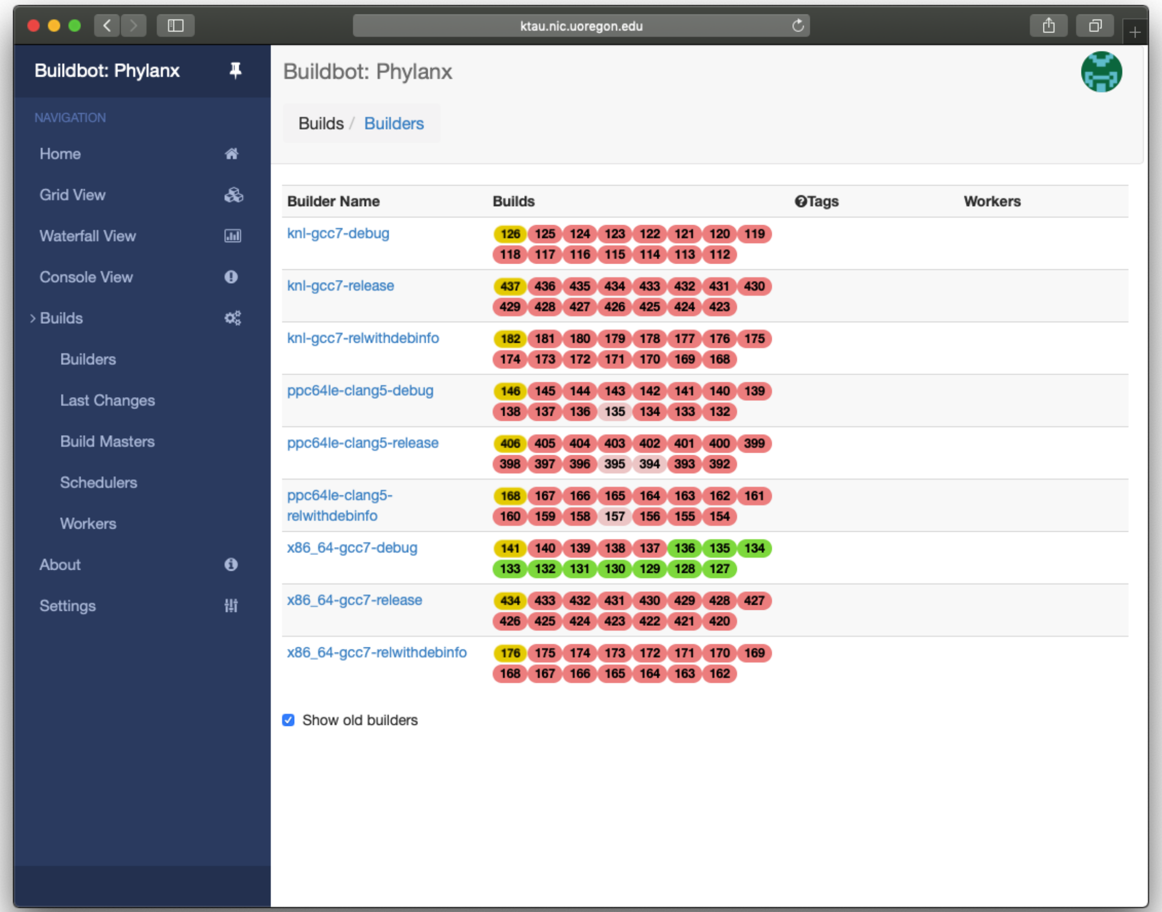
The University of Oregon team has set up a Buildbot service on a collection of compute servers hosted at the university. Buildbot is an open source framework for automating software build, test, and release processes [2]. The first server of note is the Buildbot service for testing just the APEX library. The Buildbot service monitors the Github.com APEX source code repository, and when new modifications are pushed to the repository the service launches combinations of build configurations designed to test different combinations that might reveal software errors. Figure 8 shows a screen shot of the APEX Buildbot server running on the server hosted at the University of Oregon. Both Debug and Release configurations are tested, and with different combinations of optional APEX dependencies. For the tested configurations, only one architecture (`x86_64`) is tested.



Builder Name	Builds	Tags	Workers
01-default-Debug	41 40 39 38 37 36 35		
01-default-debug	3 2 1		
01-default-Release	71 70 69 68 67 66 65 64		
01-default-release	4 3 2 1		
02-base-Debug	40 39 38 37 36 35 34		
02-base-debug	3 2 1		
02-base-Release	53 52 51 50 49 48		
02-base-release	3 2 1		
03-malloc-Debug	40 39 38 37 36 35 34		
03-malloc-debug	3 2 1		
03-malloc-Release	53 52 51 50 49 48		
03-malloc-release	3 2 1		
04-bfd-Debug	40 39 38 37 36 35 34		
04-bfd-debug	3 2 1		
04-bfd-Release	53 52 51 50 49 48 47		

Figure 8: Screen capture of the APEX Buildbot server showing a subset of build configurations.

For the Phylanx project, the University of Oregon is also hosting a Buildbot service. The Phylanx project has a number of software dependencies (all of which also have software dependencies), so the service is configured to monitor both the HPX and Phylanx source code repositories. When new code is pushed to those repositories (after having gone through an automated CI process configured for those repositories), the UO Buildbot service will check out the latest code for Phylanx, HPX, Blaze, BlazeTensor, and Pybind (all of which are software dependencies of the project). Each component will be configured and compiled, and then the final build of Phylanx will run the full suite of unit tests. Only if the entire process completes error-free will the build be considered successful. For the full integration build tests, three different architectures are tested – Intel Broadwell (`x86_64`), IBM POWER8 (`ppc64le`), and Intel MIC Knights Landing (`knl`). The `x86_64` and `knl` builds use GNU 7.1 compilers, whereas the `ppc64le` build uses LLVM/Clang 7.0 (formerly 5.0) compilers. This combination provides a broad suite of configurations, and due to the fluid nature of software development, rarely results in successful builds. Figures 9 and 10 show the Phylanx Buildbot server in action. Regardless of build success, an IRC channel monitored by the Phylanx team are notified, the web status is updated, and developers are emailed results. Monitoring the Buildbot results has found many software defects that would otherwise gone unnoticed until potentially triggered by a user.

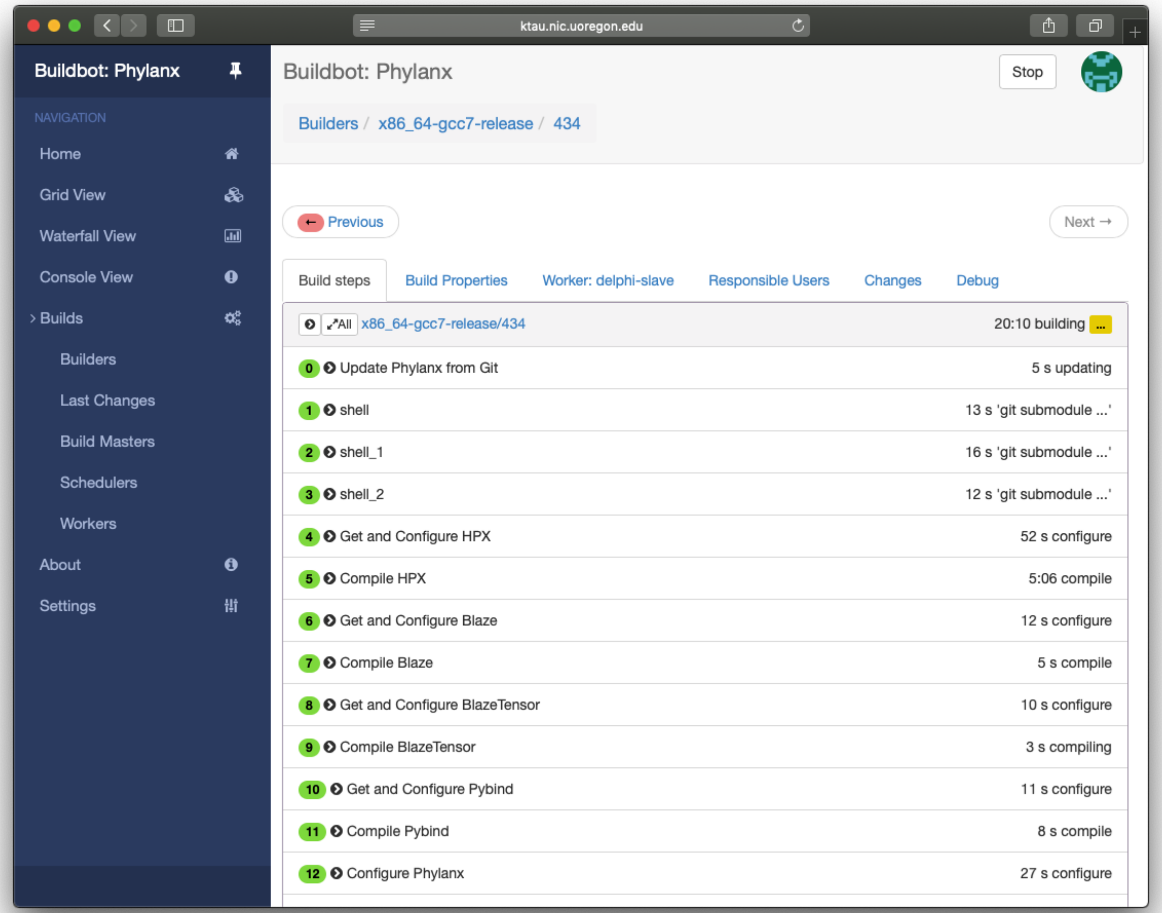


Builder Name	Builds	Tags	Workers
knl-gcc7-debug	126 125 124 123 122 121 120 119 118 117 116 115 114 113 112		
knl-gcc7-release	437 436 435 434 433 432 431 430 429 428 427 426 425 424 423		
knl-gcc7-relwithdebinfo	182 181 180 179 178 177 176 175 174 173 172 171 170 169 168		
ppc64le-clang5-debug	146 145 144 143 142 141 140 139 138 137 136 135 134 133 132		
ppc64le-clang5-release	406 405 404 403 402 401 400 399 398 397 396 395 394 393 392		
ppc64le-clang5-relwithdebinfo	168 167 166 165 164 163 162 161 160 159 158 157 156 155 154		
x86_64-gcc7-debug	141 140 139 138 137 136 135 134 133 132 131 130 129 128 127		
x86_64-gcc7-release	434 433 432 431 430 429 428 427 426 425 424 423 422 421 420		
x86_64-gcc7-relwithdebinfo	176 175 174 173 172 171 170 169 168 167 166 165 164 163 162		

Figure 9: Screen capture of Buildbot server executing a build after code commit to the Phylanx code repository.

7.2 Performance Regression Testing

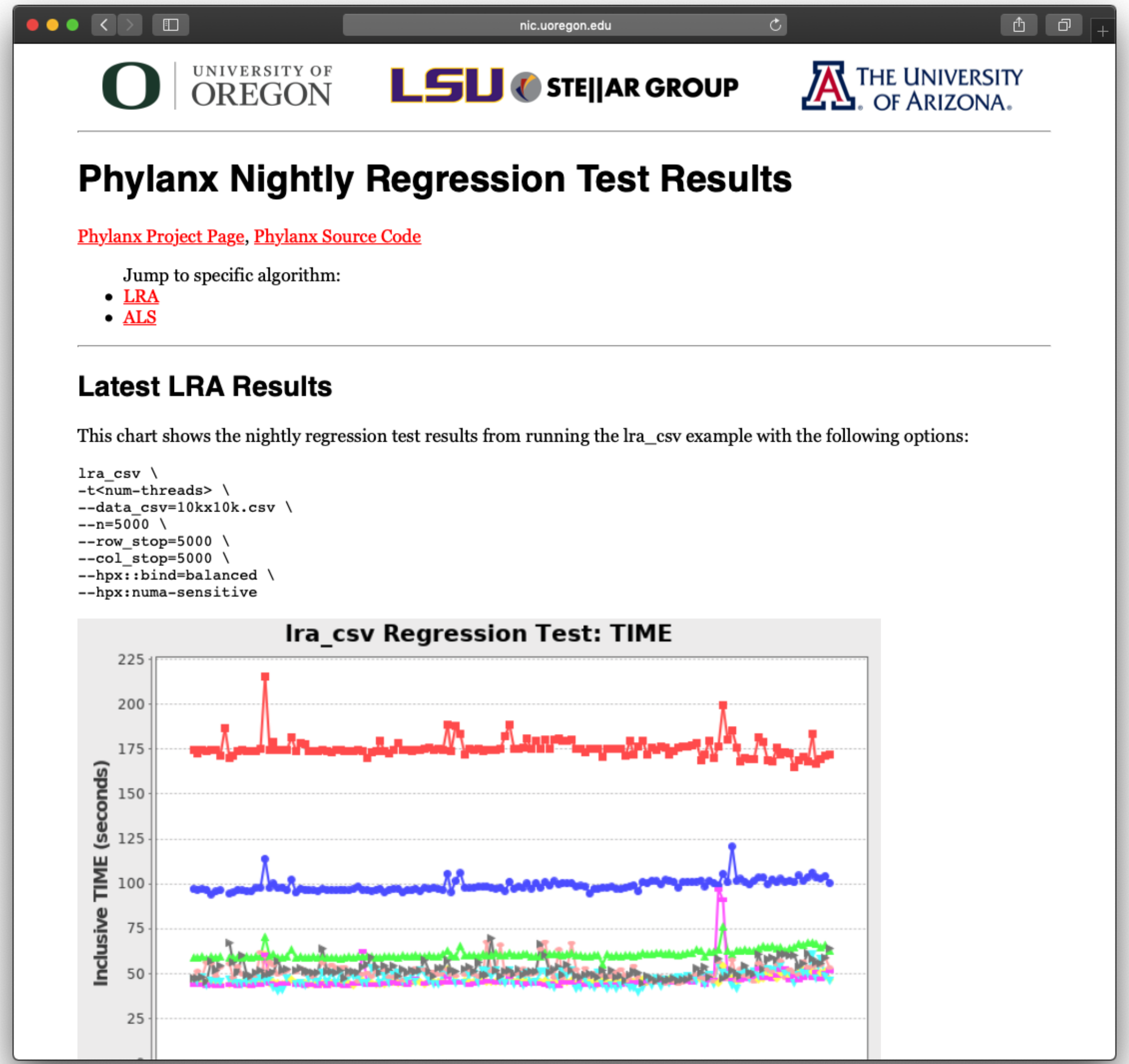
Because Phylanx is intended to be a highly performant library, the development team needs a way to confirm that software modifications do not degrade the overall performance. For that reason, the University of Oregon has configured and hosted a nightly performance regression test. Implemented with a handful of Linux shell scripts launched as a `cron` job at midnight, the Phylanx code base and all software dependencies are downloaded, configured, built and then run through a parametric test suite to capture performance data. Phylanx



All x86_64-gcc7-release/434	20:10 building
0 Update Phylanx from Git	5 s updating
1 shell	13 s 'git submodule ...'
2 shell_1	16 s 'git submodule ...'
3 shell_2	12 s 'git submodule ...'
4 Get and Configure HPX	52 s configure
5 Compile HPX	5:06 compile
6 Get and Configure Blaze	12 s configure
7 Compile Blaze	5 s compile
8 Get and Configure BlazeTensor	10 s configure
9 Compile BlazeTensor	3 s compiling
10 Get and Configure Pybind	11 s configure
11 Compile Pybind	8 s compile
12 Configure Phylanx	27 s configure

Figure 10: Screen capture of Buildbot server details while executing a build for the x86_64 platform using GCC 7 compilers.

and HPX are configured with APEX support, and at runtime the TAU Performance System is used to capture performance profiles of the applications. The profiles are stored in the TAUdb database, and using an automated analysis script, the PerfExplorer application will generate a sequence of analysis graphs including the historical performance of each tested algorithm as well as scaling charts for each algorithm. There is even a runtime breakdown of the algorithms, showing where time is spent in either HPX tasks measured by APEX or the HPX runtime as measured by TAU event-based sampling. Figure 11 shows a screen capture of the Phylanx nightly regression server, with results dating back to early September, 2018. Figure 12 shows the benefit of the performance regression testing. Some time around January 4, 2019, a performance bug was introduced into (or exposed in) the ALS algorithm. Subsequent analysis using the APEX and TAU performance measurement tools and the Traveler-Gantt and Vampir trace analysis tools showed that a modification to HPX caused idle threads to go to sleep and wait to be notified when new work is available. Unfortunately, the sleeping threads were not awakened in a timely manner. Figure 13 shows some visual evidence of the threads not waking up, in Vampir (zoomed in to show about 6 seconds of time). The top trace timeline (white background) is with the idle backoff enabled, the bottom trace (blue background) is with the idle backoff disabled. You can clearly see that the algorithm is dominated with fork-join behavior, and with the idle backoff the threads are not continuing their work in a timely manner. This particular slowdown was the result of another performance bug that has yet to be fixed - the region of low concurrency in the algorithm is an instance where the Blaze library is performing a sequential matrix multiply operation (that take about .4 seconds *each*) instead of a parallel one. We are still investigating that performance bug.



Phylanx Nightly Regression Test Results

Phylanx Project Page, Phylanx Source Code

Jump to specific algorithm:
- LRA
- ALS

Latest LRA Results

This chart shows the nightly regression test results from running the lra_csv example with the following options:

```
lra_csv \
-t<num-threads> \
--data_csv=10kx10k.csv \
--n=5000 \
--row_stop=5000 \
--col_stop=5000 \
--hpx::bind=balanced \
--hpx:numa-sensitive
```

Figure 11: Screen capture of the Phylanx nightly regression testing result web server.

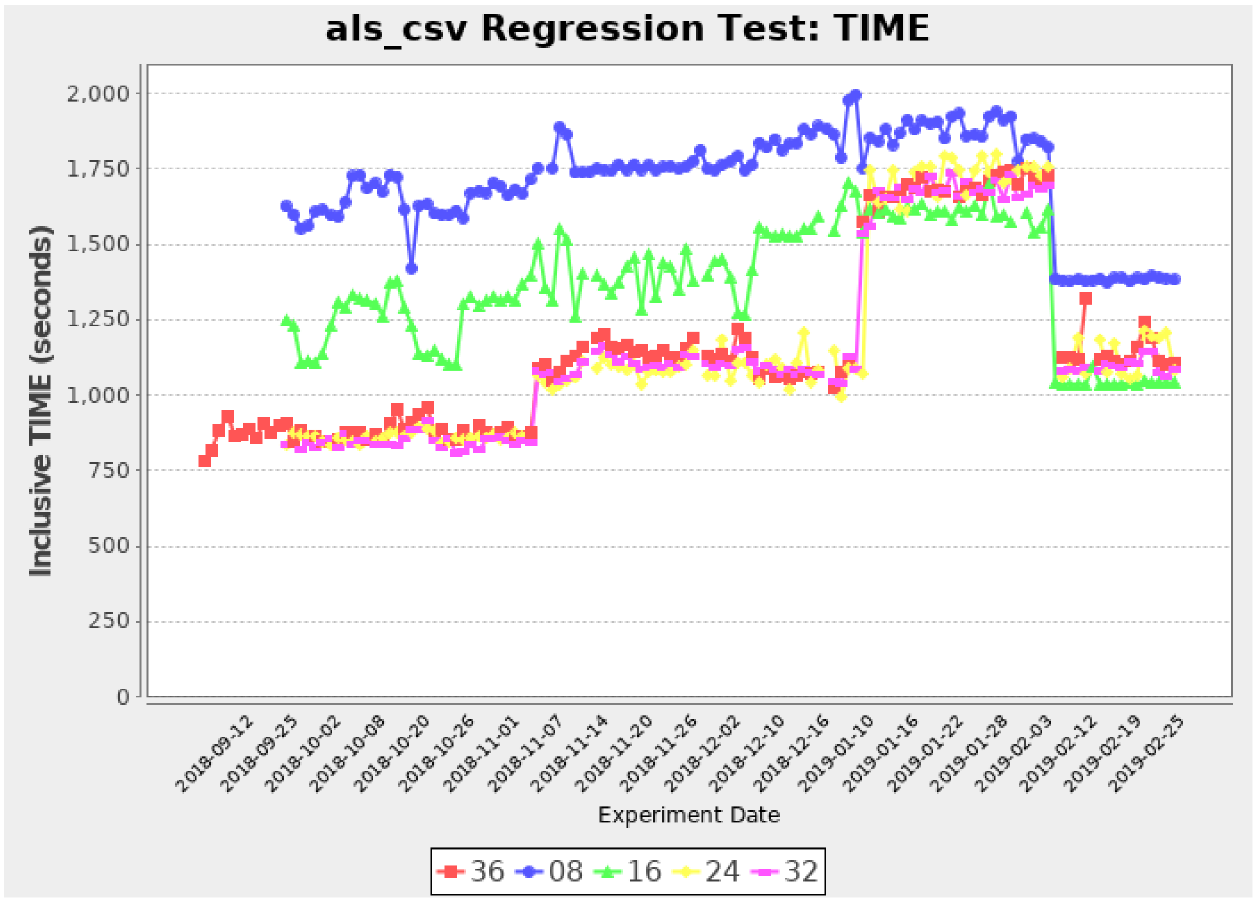


Figure 12: Performance trend for the ALS application, showing the performance bug exposed near the end of December 2018, and resolved near the end of January, 2019.

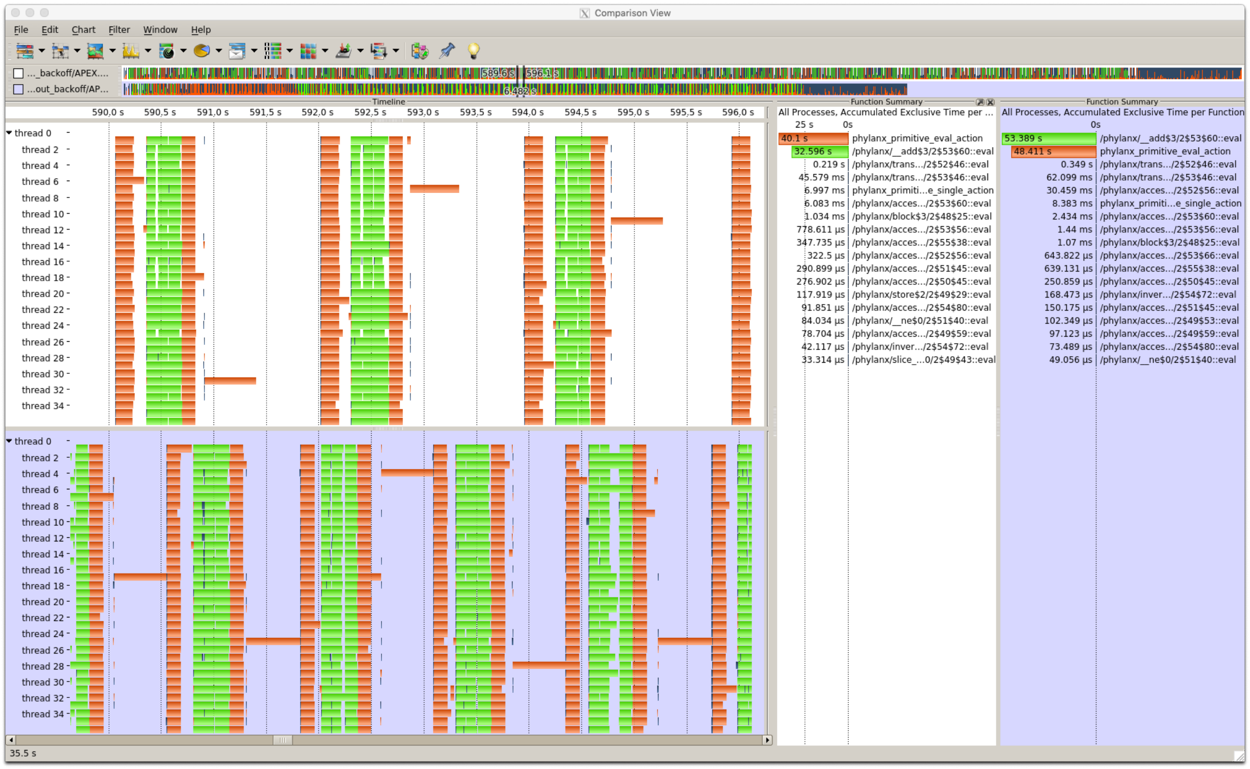
References

[1] Md Abdullah Shahneous Bari, Nicholas Chaimov, Abid M Malik, Kevin A Huck, Barbara Chapman, Allen D Malony, and Osman Sarood. Arcs: Adaptive runtime configuration selection for power-constrained openmp applications. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 461–470. IEEE, 2016.

[2] Dustin J. Mitchell Brian Warner. Buildbot : The Continuous Integration Framework, 2019. https://buildbot.net, Accessed: 2019-02-25.

[3] Thomas Heller, Bryce Adelstein Lelbach, Kevin A Huck, John Biddiscombe, Patricia Grubel, Alice E Koniges, Matthias Kretz, Dominic Marcello, David Pfander, Adrian Serio, et al. Harnessing billions of tasks for a scalable portable hydrodynamic simulation of the merger of two stars. *The International Journal of High Performance Computing Applications*, page 1094342018819744, 2019.

[4] Kevin A Huck, Allan Porterfield, Nick Chaimov, Hartmut Kaiser, Allen D Malony, Thomas Sterling, and Rob Fowler. An autonomic performance environment for exascale. *Supercomputing frontiers and innovations*, 2(3):49–66, 2015.



Function Summary

All Processes, Accumulated Exclusive Time per ...

25 s 0s	
40.1 s	phylanx_primitive_eval_action
32.596 s	/phylanx/__add$3/2$53$60::eval
0.219 s	/phylanx/trans.../2$52$46::eval
45.579 ms	/phylanx/trans.../2$53$46::eval
6.997 ms	phylanx_primiti...e_single_action
6.083 ms	/phylanx/acces.../2$53$60::eval
1.034 ms	/phylanx/block$3/2$48$25::eval
778.611 µs	/phylanx/acces.../2$53$56::eval
347.735 µs	/phylanx/acces.../2$55$38::eval
322.5 µs	/phylanx/acces.../2$52$56::eval
290.899 µs	/phylanx/acces.../2$51$45::eval
276.902 µs	/phylanx/acces.../2$50$45::eval
117.919 µs	/phylanx/store$2/2$49$29::eval
91.851 µs	/phylanx/acces.../2$54$80::eval
84.034 µs	/phylanx/__ne$0/2$51$40::eval
78.704 µs	/phylanx/acces.../2$49$59::eval
42.117 µs	/phylanx/inver.../2$54$72::eval
33.314 µs	/phylanx/slice_...0/2$49$43::eval

Function Summary

All Processes, Accumulated Exclusive Time per Function

0s	
53.389 s	/phylanx/__add$3/2$53$60::eval
48.411 s	phylanx_primitive_eval_action
0.349 s	/phylanx/trans.../2$52$46::eval
62.099 ms	/phylanx/trans.../2$53$46::eval
30.459 ms	/phylanx/acces.../2$52$56::eval
8.383 ms	phylanx_primiti...e_single_action
2.434 ms	/phylanx/acces.../2$53$60::eval
1.44 ms	/phylanx/acces.../2$53$56::eval
1.07 ms	/phylanx/block$3/2$48$25::eval
643.822 µs	/phylanx/acces.../2$53$66::eval
639.131 µs	/phylanx/acces.../2$55$38::eval
250.859 µs	/phylanx/acces.../2$50$45::eval
168.473 µs	/phylanx/inver.../2$54$72::eval
150.175 µs	/phylanx/acces.../2$51$45::eval
102.349 µs	/phylanx/acces.../2$49$53::eval
97.123 µs	/phylanx/acces.../2$49$59::eval
73.489 µs	/phylanx/acces.../2$54$80::eval
49.056 µs	/phylanx/__ne$0/2$51$40::eval

Figure 13: Before-and-after views of the Phylanx ALS application as captured by APEX and visualized in Vampir.

[5] Katherine E Isaacs, Abhinav Bhatele, Jonathan Lifflander, David Böhme, Todd Gamblin, Martin Schulz, Bernd Hamann, and Peer-Timo Bremer. Recovering logical structure from charm++ event traces. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.

[6] Katherine E Isaacs, Peer-Timo Bremer, Ilir Jusufi, Todd Gamblin, Abhinav Bhatele, Martin Schulz, and Bernd Hamann. Combing the communication hairball: Visualizing parallel execution traces using logical time. *IEEE transactions on visualization and computer graphics*, 20(12):2349–2358, 2014.

[7] R Tohid, Bibek Wagle, Shahrzad Shirzad, Patrick Diehl, Adrian Serio, Alireza Kheirkhahan, Parsa Amini, Katy Williams, Kate Isaacs, Kevin Huck, et al. Asynchronous execution of python code on task based runtime systems. *arXiv preprint arXiv:1810.07591*, 2018.

[8] Bibek Wagle, Samuel Kellar, Adrian Serio, and Hartmut Kaiser. Methodology for adaptive active message coalescing in task based runtime systems. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1133–1140. IEEE, 2018.

A Web Links to Relevant Technology

1. *Phylanx – A Distributed Array Toolkit*, The project web page for the Phylanx project
http://phylanx.stellar-group.org

2. *Phylanx source code*, The code repository for the Phylanx project
https://github.com/STEllAR-GROUP/phylanx

3. *HPX – High Performance ParalleX*, The project web page for the HPX project
http://stellar-group.org/libraries/hpx

4. *HPX source code*, The code repository for the HPX project, a build dependency of Phylanx
https://github.com/STEllAR-GROUP/hpx

5. *APEX source code*, The code repository for APEX, an optional build dependency of HPX
https://github.com/khuck/xpress-apex

6. *OTF2 source code*, The project web page for OTF2, an optional build dependency of APEX
https://www.vi-hps.org/projects/score-p/

7. *Active Harmony source code*, The project web page for Active Harmony, an optional build dependency of APEX
https://www.dyninst.org/harmony

8. *TAU – Tuning and Analysis Utilities*, The project web page for the TAU Performance System, a profiling and tracing toolkit integrated with APEX
http://tau.uoregon.edu

9. *TAU source code* Public mirror of the TAU source code
https://github.com/UO-OACISS/tau2

10. *Traveler-Gantt source code*, The code repository for Traveler-Gantt, an OTF2 visualization tool for APEX traces of HPX applications
https://github.com/hdc-arizona/traveler-gantt

11. *APEX Buildbot Server*, The web address of the APEX Buildbot server
http://ktau.nic.uoregon.edu:8010

12. *APEX Buildbot Scripts source code*, The scripts used to build Phylanx and all its dependencies for the Buildbot services
https://github.com/khuck/phylanx-buildbot-scripts

13. *Phylanx Buildbot Server*, The web address of the Phylanx Buildbot server
http://ktau.nic.uoregon.edu:8020

14. *Buildbot Source Code*,
https://buildbot.net

15. *Phylanx Nightly Regression Tests*, The web address for the Phylanx nightly regression test results
http://www.nic.uoregon.edu/ khuck/regression/phylanx

B Code Listings

```
//  Copyright (c) 2007-2016 Hartmut Kaiser
//
//  Distributed under the Boost Software License, Version 1.0. (See
    accompanying
//  file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

#pragma once // prevent multiple inclusions of this header file.

#include <hpx/config.hpp>
#include <hpx/runtime/get_locality_id.hpp>
#include <hpx/util/thread_description.hpp>
#include <hpx/runtime/get_num_localities.hpp>
#include <hpx/runtime/startup_function.hpp>

#ifdef HPX_HAVE_APEX
#include "apex_api.hpp"
#include <memory>
#include <cstdint>
#include <string>
typedef std::shared_ptr<apex::task_wrapper> apex_task_wrapper;
#else
typedef void* apex_task_wrapper;
#endif

namespace hpx { namespace util
{
#ifdef HPX_HAVE_APEX

    static void hpx_util_apex_init_startup(void)
    {
        apex::init(nullptr, hpx::get_locality_id(),
            hpx::get_initial_num_localities());
    }

    inline void apex_init()
    {
        hpx_util_apex_init_startup();
        //hpx::register_pre_startup_function(&hpx_util_apex_init_startup);
    }

    inline void apex_finalize()
    {
        apex::finalize();
    }

    HPX_EXPORT apex_task_wrapper apex_new_task(
                 thread_description const& description,
                 std::uint32_t parent_task_locality,
                 threads::thread_id_type const& parent_task);

    inline apex_task_wrapper apex_update_task(apex_task_wrapper wrapper,
                 thread_description const& description)
    {
        if (wrapper == nullptr) {
            threads::thread_id_type parent_task(nullptr);
            // doesn't matter which locality we use, the parent is null
            return apex_new_task(description, 0, parent_task);
        } else if (description.kind() == thread_description::
   data_type_description) {
            return apex::update_task(wrapper,
                description.get_description());
        } else {
            return apex::update_task(wrapper,
                description.get_address());
        }
    }

    inline apex_task_wrapper apex_update_task(apex_task_wrapper wrapper,
   char const* name)
    {
        if (wrapper == nullptr) {
            apex_task_wrapper parent_task(nullptr);
            return apex::new_task(std::string(name), UINTMAX_MAX,
   parent_task);
        }
        return apex::update_task(wrapper, name);
    }

    /* This is a scoped object around task scheduling to measure the time
     * spent executing hpx threads */
    struct apex_wrapper
    {
        apex_wrapper(apex_task_wrapper data_ptr) : stopped(false), data_(
   nullptr)
        {
            /* APEX internal actions are not timed.  Otherwise, we would
             * end up with recursive timers. So it's possible to have
             * a null task wrapper pointer here. */
            if (data_ptr != nullptr) {
                data_ = data_ptr;
                apex::start(data_);
            }
        }
        ~apex_wrapper()
        {
            stop();
        }

        void stop() {
            if(!stopped) {
                stopped = true;
            /* APEX internal actions are not timed.  Otherwise, we would
             * end up with recursive timers. So it's possible to have
             * a null task wrapper pointer here. */
                if (data_ != nullptr) {
                    apex::stop(data_);
                }
            }
        }

        void yield() {
            if(!stopped) {
                stopped = true;
            /* APEX internal actions are not timed.  Otherwise, we would
             * end up with recursive timers. So it's possible to have
             * a null task wrapper pointer here. */
                if (data_ != nullptr) {
                    apex::yield(data_);
                }
            }
        }

        bool stopped;
        apex_task_wrapper data_;
    };

    struct apex_wrapper_init
    {
        apex_wrapper_init(int /*argc*/, char ** /*argv*/)
        {
            //apex::init(nullptr, hpx::get_locality_id(),
            //    hpx::get_initial_num_localities());
            hpx::register_pre_startup_function(&hpx_util_apex_init_startup
   );
        }
        ~apex_wrapper_init()
        {
            apex::finalize();
        }
    };
#else
    inline void apex_init() {}
    inline void apex_finalize() {}

    inline apex_task_wrapper apex_new_task(
                thread_description const& description,
                std::uint32_t parent_task_locality,
                threads::thread_id_type const& parent_task) {return
   nullptr;}

    inline apex_task_wrapper apex_update_task(apex_task_wrapper wrapper,
                thread_description const& description) {return nullptr;}

    inline apex_task_wrapper apex_update_task(apex_task_wrapper wrapper,
                 char const* name) {return nullptr;}

    struct apex_wrapper
    {
        apex_wrapper(apex_task_wrapper data_ptr) {}
        ~apex_wrapper() {}
        void stop(void) {}
        void yield(void) {}
    };

    struct apex_wrapper_init
    {
        apex_wrapper_init(int argc, char **argv) {}
        ~apex_wrapper_init() {}
    };
#endif
}}
```

Listing 10: hpx/util/apex.hpp

```
//  Copyright (c) 2007-2013 Kevin Huck
//
//  Distributed under the Boost Software License, Version 1.0. (See
   accompanying
//  file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)
//

#include <hpx/config.hpp>
#include <hpx/util/apex.hpp>
#include <hpx/runtime/threads/thread_helpers.hpp>
#include <hpx/runtime/threads/thread_data.hpp>
#include <hpx/runtime/find_localities.hpp>
#include <cstdint>

namespace hpx { namespace util
{
#ifdef HPX_HAVE_APEX
    apex_task_wrapper apex_new_task(
            thread_description const& description,
            std::uint32_t parent_locality_id,
            threads::thread_id_type const& parent_task)
    {
        static std::uint32_t num_localities = hpx::
   get_initial_num_localities();
        apex_task_wrapper parent_wrapper = nullptr;
        // Parent pointers aren't reliable in distributed runs.
        if (parent_task != nullptr &&
            num_localities == 1
            /*hpx::get_locality_id() == parent_locality_id*/) {
            parent_wrapper = parent_task.get()->get_apex_data();
        }
        if (description.kind() ==
                thread_description::data_type_description) {
            return apex::new_task(description.get_description(),
                    UINTMAX_MAX, parent_wrapper);
        } else {
            return apex::new_task(description.get_address(),
                    UINTMAX_MAX, parent_wrapper);
        }
    }

#endif
}}
```
Listing 11: hpx/util/apex.cpp